

Intelligent Agents

Planning Graphs - The Graph Plan Algorithm

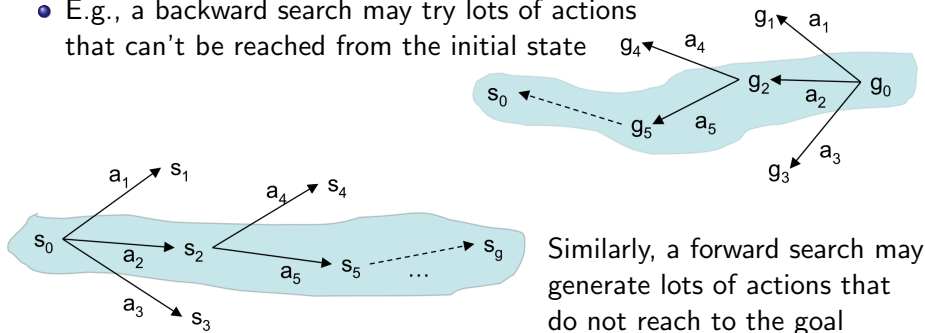
Ute Schmid

Cognitive Systems, Applied Computer Science, Bamberg University

last change: June 18, 2014

Motivation

- A big source of inefficiency in search algorithms is the *branching factor*
 - ◊ the number of children of each node
- E.g., a backward search may try lots of actions that can't be reached from the initial state



Similarly, a forward search may generate lots of actions that do not reach to the goal

One way to reduce branching factor

- First create a relaxed problem
 - ◇ Remove some restrictions of the original problem
 - ▷ Want the relaxed problem to be easy to solve (polynomial time)
 - ◇ The solutions to the relaxed problem will include all solutions to the original problem
- Then do a modified version of the original search
 - ◇ Restrict its search space to include only those actions that occur in solutions to the relaxed problem

Outline

- The Graphplan algorithm
- Planning graphs
 - ◇ example
- Mutual exclusion
 - ◇ example (continued)
- Doing solution extraction
 - ◇ example (continued)
- Discussion
- Extract heuristic values from planning graph
- FF-plan

Graphplan

procedure Graphplan:

- for $k = 0, 1, 2, \dots$

- ◊ *Graph expansion:*

- ▷ create a "planning graph" that contains k "levels"

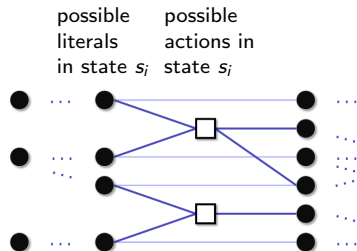
- ◊ Check whether the planning graph satisfies a necessary (but insufficient) condition for plan existence

- ◊ If it does, then

- ▷ *do solution extraction:*

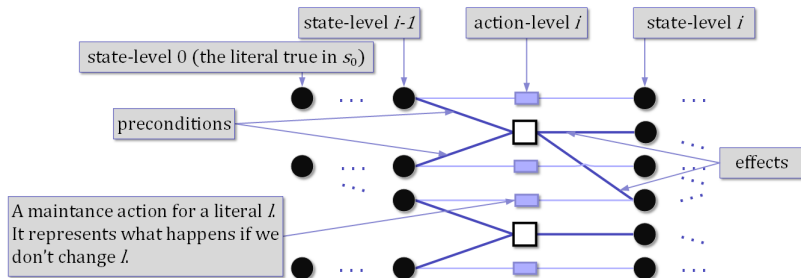
- backward search, modified to consider only the actions in the planning graph
 - if we find a solution, then return it

relaxed
problem



The Planning Graph

- Search space for a relaxed version of the planning problem
- Alternating layers of ground literals and actions
 - Nodes at action-level i : actions that might be possible to execute at time i
 - Nodes at state-level i : literals that might possibly be true at time i
 - Edges: preconditions and effects



Example

- Due to Dan Weld (U. of Washington)
- Suppose you want to prepare dinner as a surprise for your sweetheart (who is asleep)

$$s_0 = \{\text{garbage, cleanHands, quiet}\}$$

$$g = \{\text{dinner, present, } \neg \text{garbage}\}$$

<u>Action</u>	<u>Preconditions</u>	<u>Effects</u>
cook()	cleanHands	dinner
wrap()	quiet	present
carry()	none	\neg garbage, \neg cleanHands
dolly()	none	\neg garbage, \neg quiet

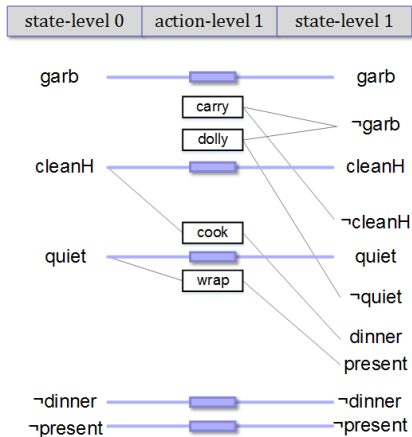
Also have the maintenance action: one for each literal

Example (continued)

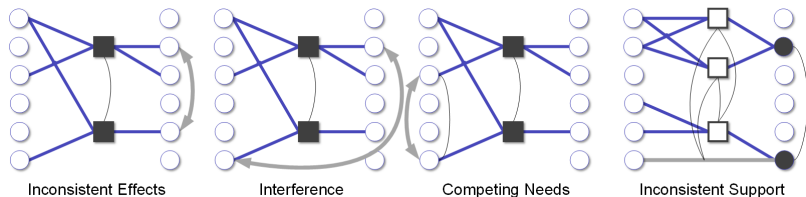
- state-level 0:
 - $\{\text{all atoms in } s_0\} \cup$
 - $\{\text{negations of all atoms not in } s_0\}$
- action-level 1:
 - $\{\text{all actions whose preconditions}$
 - $\text{are satisfied and non-mutex in } s_0\}$
- state-level 1:
 - $\{\text{all effects of all of the}$
 - $\text{actions in action-level 1}\}$

<u>Action</u>	<u>Preconditions</u>	<u>Effects</u>
cook()	cleanHands	dinner
wrap()	quiet	present
carry()	none	\neg garbage, \neg cleanHands
dolly()	none	\neg garbage, \neg quiet

Also have the maintenance action



Mutual Exclusion



- Two actions at the same action-level are mutex if
 - ◇ *Inconsistent effects*: an effect of one negates an effect of the other
 - ◇ *Interference*: one deletes a precondition of the other
 - ◇ *Competing needs*: **they have mutually exclusive preconditions**
- Otherwise they don't interfere with each other
 - ◇ Both may appear in a solution plan
- Two literals at the same state-level are mutex if
 - ◇ *Inconsistent support*: one is the negation of the other, **or all ways of achieving them are pairwise mutex**

Recursive propagation of mutexes

Example (continued)

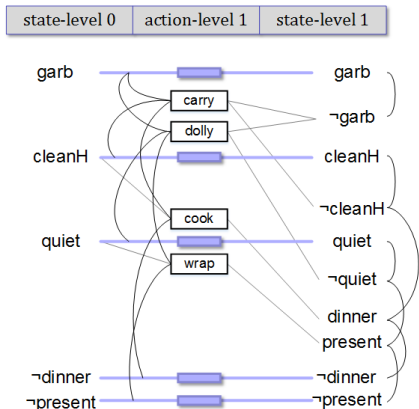
- Augment the graph to indicate mutexes
 - *carry* is mutex with the maintenance action for garbage (inconsistent effects)
 - *dolly* is mutex with *wrap* interference
 - \sim *quiet* is mutex with *present* inconsistent support
- each of *cook* and *wrap* is mutex with a maintenance operation

<u>Action</u>	<u>Preconditions</u>	<u>Effects</u>
cook()	cleanHands	dinner
wrap()	quiet	present
carry()	<i>none</i>	\neg garbage, \neg cleanHands
dolly()	<i>none</i>	\neg garbage, \neg quiet

Also have the maintenance action

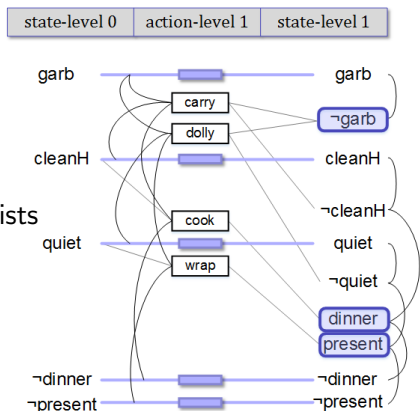
Dana Nau: Lecture slides for *Automated Planning*

Licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License: <http://creativecommons.org/licenses/by-nc-sa/2.0/>



Example (continued)

- Check to see whether there's a possible solution
- Recall that the goal is
 - ◊ $\{\neg\textit{garbage}, \textit{dinner}, \textit{present}\}$
- Note that in state-level 1,
 - ◊ All of them are there
 - ◊ None are mutex with each other
- Thus, there's a chance that a plan exists
- Try to find it
 - Solution extraction



Recall what the algorithm does

procedure Graphplan:

- for $k = 0, 1, 2, \dots$
 - ◇ *Graph expansion*:
 - ▷ create a "planning graph" that contains k "levels"
 - ◇ Check whether the planning graph satisfies a necessary (but insufficient) condition for plan existence
 - ◇ If it does, then
 - ▷ *do solution extraction*:
 - backward search, modified to consider only the actions in the planning graph
 - if we find a solution, then return it

Solution Extraction

The set of goals we are trying to achieve

The level of the state s_i

procedure Solution-extraction(g, i)

if $i=0$ then return the solution
for each literal l in g

non-deterministically choose an action to use in state s_{i-1} to achieve l

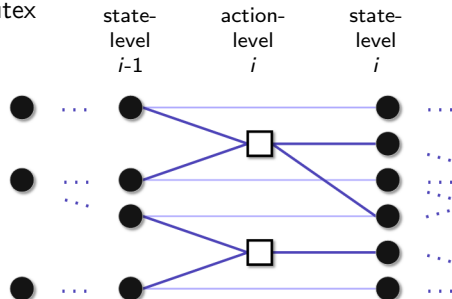
if any pair of chosen actions are mutex
then backtrack

$g' := \{ \text{the preconditions of the chosen actions} \}$

Solution-extraction($g', i-1$)

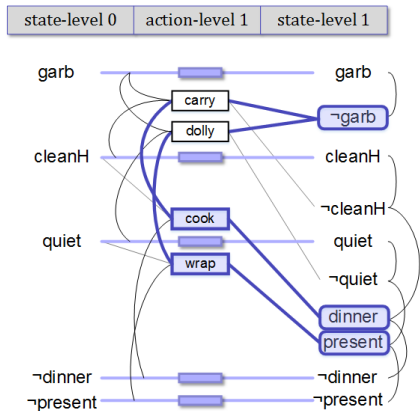
end Solution-extraction

A real action or a maintenance action



Example (continued)

- Two sets of actions for the goals at state-level 1
- Neither of them works
 - ◊ Both sets contain actions that are mutex

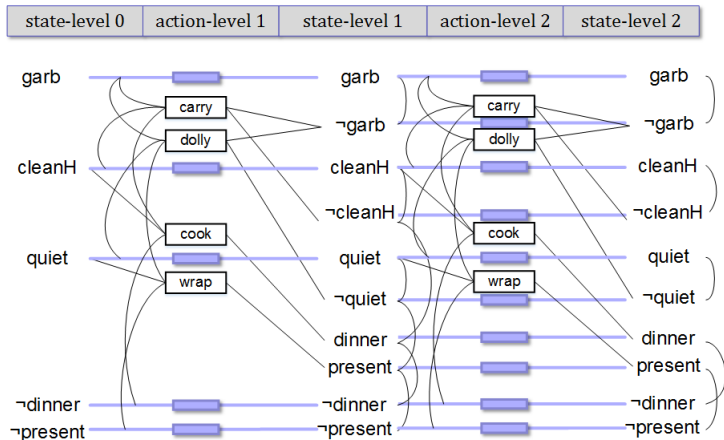


Recall what the algorithm does

procedure Graphplan:

- for $k = 0, 1, 2, \dots$
 - ◇ *Graph expansion*:
 - ▷ create a "planning graph" that contains k "levels"
 - ◇ Check whether the planning graph satisfies a necessary (but insufficient) condition for plan existence
 - ◇ If it does, then
 - ▷ *do solution extraction*:
 - backward search, modified to consider only the actions in the planning graph
 - if we find a solution, then return it

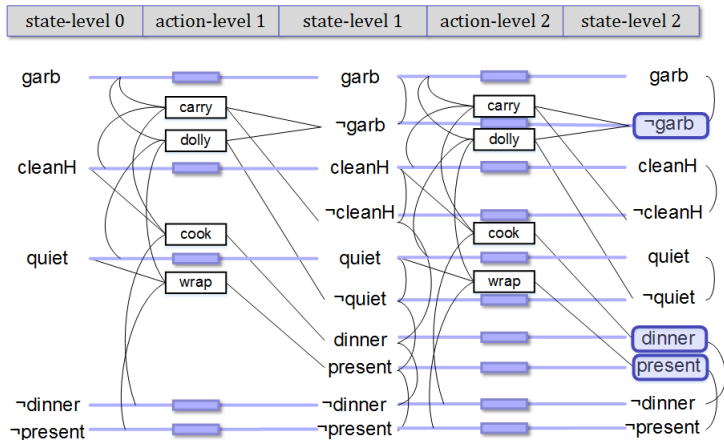
Example (continued)



- Go back and do more graph expansion

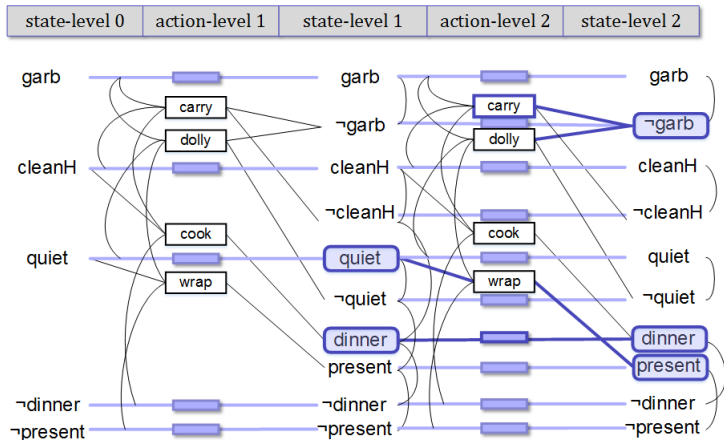
- Generate another action-level and another state-level

Example (continued)



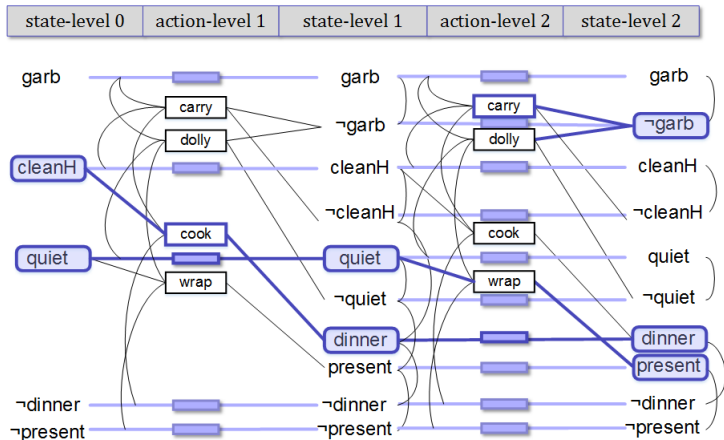
- Solution extraction
- Twelve combinations at level 4
 - Three ways to achieve \neg garb
 - Two ways to achieve dinner
 - Two ways to achieve present

Example (continued)



- Several of the combinations look OK at level 2
- Here's one of them

Example (continued)



- Call Solution-Extraction recursively at level 2
- It succeeds
- Solution whose parallel length is 2

Properties of GraphPlan

- GraphPlan is sound and complete
 - If Graphplan returns a plan, then that plan is a solution to the planning problem
 - If there are solutions to the planning problem, then GraphPlan returns one of them
- The size of the planning graph GraphPlan generates is polynomial in the size of the planning problems
- The planning algorithm always terminates
 - There is a fix-point on the number of levels of the planning graphs such that the algorithm either generates a solution or returns failure
- GraphPlan is a **partial order** planner
 - Actions located at the same level which are not mutex are given as "simultaneously"
 - A total order plan can be generated by constructing an arbitrary sequence from the parallel actions

History

- **GraphPlan** was the first planner that used planning-graph techniques
- Before GraphPlan came out, most planning researchers were working on PSP-like planners
 - POP, SNLP, UCPOP, etc.
- The size of the planning graph GraphPlan generates is polynomial in the size of the planning problems
- GraphPlan caused a sensation because it was so much faster
- Many subsequent planning systems have used ideas from it
 - IPP, STAN, GraphHTN, SGP, Blackbox, Medic, TGP, LPG
 - Many of them are much faster than the original Graphplan

Comparison with Plan-Space Planning

- Advantage:
 - The backward-search part of Graphplan - which is the hard part - will only look at the actions in the planning graph
 - smaller search space than PSP; thus faster
- Disadvantage:
 - To generate the planning graph, Graphplan creates a huge number of ground atoms
 - Many of them may be irrelevant
- Can alleviate (but not eliminate) this problem by assigning data types to the variables and constants
 - Only instantiate variables to terms of the same data type
- For classical planning, the advantage outweighs the disadvantage
 - GraphPlan solves classical planning problems much faster than PSP

Getting Heuristic Values from a Planning Graph

- Planning graphs can be used to get heuristic values for heuristic search planning
- Recall how GraphPlan works:
loop

Graph expansion:

this takes polynomial time

extend a "planning graph" forward from the initial state until we have achieved a necessary (but insufficient) condition for plan existence

Solution extraction:

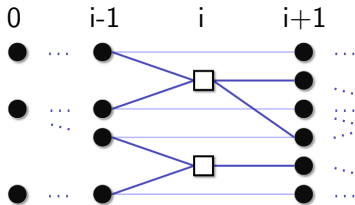
this takes exponential time

search backward from the goal, looking for a correct plan if we find one, then return it

repeat

Using Planning Graphs to Compute $h(s)$

- In the graph, there are alternating layers of ground literals and actions
- The number of "action" layers is a lower bound on the number of actions in the plan
- Construct a planning graph, starting at s
- $\Delta^g(s, p) =$ level of the first layer that "possibly achieves" p
- $\Delta^g(s, g)$ is very close to $\Delta_2(s, g)$
 - $\Delta_2(s, g)$ counts each action individually
 - $\Delta^g(s, g)$ groups together the independent actions in a layer



The FastForward Planner

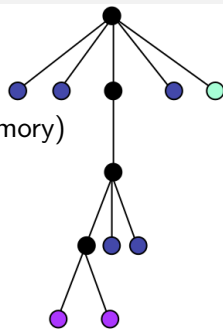
- Use a heuristic function similar to $h(s) = \Delta^g(s, g)$
 - Some ways to improve it (I'll skip the details)
- Don't want an A*-style search (takes too much memory)
- Instead, use a greedy procedure:

until we have a solution, do

expand the current state s

$s :=$ the child of s for which $h(s)$ is smallest

(i.e., the child we think is closest to a solution)



- There are some ways to improve this (I'll skip the details)
- Can't guarantee how fast it will find a solution, or how good a solution it will find
 - However, it works pretty well on many problems

AIPS-2000 Planning Competition

- *FastForward* did quite well
- In this competition, all of the planning problems were classical problems
- Two tracks:
 - "Fully automated" and "hand-tailored" planners
 - *FastForward* participated in the fully automated track
 - It got one of the two "outstanding performance" awards
 - Large variance in how close its plans were to optimal
 - However, it found them very fast compared with the other fully-automated planners

2002 International Planning Competition

- Among the automated planners, *FastForward* was roughly in the middle
- LPG (graphplan + local search) did much better, and got a "distinguished performance of the first order" award
- It's interesting to see how *FastForward* did in problems that went beyond classical planning
 - Numbers, optimization
- Example: Satellite domain, numeric version
 - A domain inspired by the Hubble space telescope (a lot simpler than the real domain, of course)
 - A satellite needs to take observations of stars
 - Gather as much data as possible before running out of fuel
 - Any amount of data gathered is a solution
 - Thus, *FastForward* always returned the null plan

2004 International Planning Competition

- *FastForward*'s author was one of the competition chairs
 - Thus FastForward did not participate

Summary

- Graphplan is an efficient algorithm for domain-independent planning
- Graphplan works in two steps: Graph expansion and solution extraction
- Introducing mutex relations helps to restrict search by eliminating not admissible combinations of literals
- Graphplan is a partial order planner: actions which are independent can be in parallel, a total order plan can be generated from the partial-order solution
- The introduction of Graphplan in 1997 was a break-through in planning research: A. Blum and M. Furst (1997). Fast Planning Through Planning Graph Analysis. <<http://www.cs.cmu.edu/%7Eavrim/Papers/graphplan.ps>>/ArtificialIntelligence/, 90:281--300
- Graphplan was inspired by dynamic programming algorithms, especially dealing with network flow problems
- Graphplan was followed by many new algorithms which either built on Graphplan or proposed applications of other efficient algorithms to planning (e.g., SAT-Planning)